I/O on the Tera File System

Tera Computer Company
400 N. 34th St.
Seattle, WA 98103

February 5, 1993

1 Introduction

This document describes the mechanism for performing input and output on the Tera File System. In particular, we discuss the functional characteristics of the *read()*, *write()*, *mmap()*, and *munmap()* system calls as they are implemented on Tera.

2 Read and Write

A *read()* system call reads a specified number of bytes from a file's data blocks into a buffer location in the user's virtual address space. Similarly, a *write()* system call writes a specified number of bytes from a user's virtual addressed buffer into a file's data blocks.

Currently, there are two different approaches to implement *read()* and *write()*. We will discuss and evaluate them in the following sections.

2.1 Implementation 1

A conventional way to implement *fread()* and formatted *reads* involves a call to a library routine that dispenses data from its own internal buffer until it runs out, at which time it does a system call, *sys_read()* to the file system. The file system moves via *uiomove()* a specified amount of data from its buffer cache into the library buffer. This continues until the requested amount of data has been read into a user supplied buffer. In this scenario, the size of the library buffer determines the frequency of system calls made to the file system.

Using *fread()* as an example, the following pseudocode illustrates a conventional *Unix* implementation of *fread()*. Note that this example is not intended to be the actual *Unix* code. This pseudocode can be made thread-safe by parallelizing accesses to the library buffer. A barrier must be provided for all executing threads each time new data is read into the library buffer.

```
user code:
----------
    FILE *stream;
    char *buf;
    int len;

    stream = fopen(filename, "r");
    error = fread(buf, len, 1, stream);

library code:
-------------
fread(stream, buf, len) {
    char *libbuf = stream->_p; /* library buffer for stream */
    int resid = stream->_r; /* data not yet read in libbuf */

    while(len) {             /* loop till actual len is read */
        if(!resid) {         /* if no more data in libbuf */
                /* call file system to read data into libbuf */
                libbuf = stream->_p = stream->_sbuf.base;
                resid = sys_read();
                /* reinitialize libbuf base address */
        }
        /* copy data from libbuf to user buf */
        bcopy(buf, libbuf, size);
        libbuf += size;    /* increment libbuf ptr */
        /* calculate what has not been read in libbuf */
        resid -= size;
        len -= size;
    }
    stream->_p = libbuf; /* update ptr to libbuf */
}

system code:
------------
sys_read() {
    lock_vnode();
    /* do file system and device dependent read */
    VOP_READ();
    uiomove();              /* a copy to libbuf */
    fd->f_offset += len;    /* update offset in fdesc table */
    *retval = len;
    unlock_vnode();
}
```

2.2 Implementation 2

The *mmap()* system call with a *PROT_SHARED* option has been proposed as a possible alternative for performing user *read()* and *write()*. This implementation is very similar to the previous one except data resides in a shared buffer cache memory instead of an internal library buffer area. First, a call to *munmap()* is called to ensure that any previously *mmapped* segment is unmapped from the specified virtual address range. Then, *mmap()* is called to map a block of data less than or equal to the size of a file's data block to the task's virtual address space.

Note that by using *mmap()* as a mechanism for all *reads* and *writes*, the maximum limit of opened file descriptors will be restricted by the maximum number of shared *mmapped* data blocks allowable per task. Since the limits on opened file descriptors is usually larger in comparison to the limits on shared *mmapped* data blocks, this is not a desirable codependency.

Using *fread()* again as an example, the following pseudocode shows how *mmap()* is used for implementing *fread()*. Parallelization of this code is similar to that of a conventional implementation.

```
user code:
----------

    FILE *stream;
    char *buf;
    int len;

    stream = fopen(filename, "r");
    error = fread(buf, len, 1, stream);

library code:
-------------
fread(stream, buf, len) {
    int size;
    char *mmapbuf = stream->_p; /* mmap buffer for stream */
    int resid = stream->_r; /* data not yet read in mmapbuf */

    while(len) {            /* loop till actual len is read */
        if(!resid) {        /* if no more data in mmapbuf */
                /* first, unmap current buffer */
                munmap(stream->_sbuf.base);
                /* mmap as shared */
                mmapbuf = mmap(0, &resid, prot, flag, stream->_file);
                /* reinitialize mmap buffer base address */
                stream->_p = stream->_sbuf.base = mmapbuf;
        }
        /* copy data from mmapbuf to user buf */
        bcopy(buf, mmapbuf, size);
        mmapbuf += size; /* increment mmapbuf ptr */
        /* calculate what has not been read in mmapbuf */
        resid -= size;
        len -= size;
    }
    stream->_p = mmapbuf; /* update stream ptr */
}
```

```
system code:
------------

mmap() {
    lock_vnode();
    /* do file system and device dependent read */
    VOP_READ();
    pa = vm_mmap();        /* mmap to user's addr space
                              & increment reference count */
    unlock_vnode();
    *retval = pa;
}

munmap() {
    vm_munmap(pa);         /* unmap from user's addr space
                              & decrement reference count */
}
```

2.3 Evaluations of Implementations

Table 1 compares these two approaches. The primary advantage of *mmap* is saving a data copy from system to library buffer. However, at first glance it is not clear what effect the *mmap()* implementation will have on global performance of the system; particularly, when time between *usr_read()*'s can be large, thus tying up valuable file system's buffer cache between an *mmap()* and its *munmap()*. Our plan is to stay with the conventional implementation. In the future, with performance studies we will explore the global effects of an *mmap* implementation.

3 Mmap and Munmap

An *mmap()* system call allows a user to share a file's data that resides in the file system's buffer cache by directly mapping the buffer cache block into a task's virtual address space. The advantages of sharing data between the operating system and its users are to eliminate extra copying between virtual address spaces; and to provide a means for synchronization between tasks of different address

mmap	sys_call
no copy to libbuf	copy from system to libbuf
invoke 0 to 2 syscalls	invoke 0 to 1 syscall
buffer cache tied up by user	buffer cache not controlled by user
max fd = max mmapbufs	no relations max fd and mmapbufs
access 1 FS blk per syscall	access 1 or more FS blks per syscall

TABLE 1: Compare Read and Write Using Mmap or Sys_Call

spaces. An *munmap()* system call removes the mapping of part or all of the mmapped block from a task's address space. Except for specifics discussed in this section *mmap()* and *munmap()* are intended to be SVID 3 compliant.

This document primarily describes *mmap()* as it relates to data shared using the file system's name space. For example, we will not be discussing anonymous *mmap()* for implementation of dynamic shared memory with no persistent store.

3.1 Characteristics of Mmap

A file must be opened prior to its data being *mmapped* into a user's address space.

Table 2 shows all valid combinations of flags specified in an *open()* call and its corresponding protection flags in *mmap()*.

Other general characteristics of *mmap()* that are worth mentioning include:

1. the offset of a file descriptor is not affected after an *mmap()* call
2. reference count on an *mmapped* buffer is incremented after a *fork()* system call and is decremented after an *exec()* system call
3. SVR4 allows *mmapping* over an address range that is already *mmapped*. This essentially performs an *munmap()* of the old segment and an *mmap()* of the new segment in the same address range. Because there is potential for hiding programming errors, currently we are inclined to be stricter in our functionality. On Tera an application must first unmap an existing mapped segment before another physical segment can be mapped within the same address range. Otherwise, an error (*EADDRINUSE*) will be returned to its caller.

open flag	mmap prot flag	Return
O_RDONLY	PROT_READ	OK
O_RDONLY	PROT_WRITE	EACCES
O_WRONLY	PROT_READ	EACCES
O_WRONLY	PROT_WRITE	OK
O_RDWR	PROT_READ	OK
O_RDWR	PROT_WRITE	OK
O_RDWR.O_CREAT	PROT_READ	OK
O_RDWR.O_CREAT	PROT_WRITE	OK
O_RDWR.O_APPEND	PROT_READ	OK
O_RDWR.O_APPEND	PROT_WRITE	OK

TABLE 2: Validity of Open and Mmap Flags

3.2 Mmap on Tera

Architectural differences between the memory models of the Tera Computer System [1] and most conventional page based systems are manifested in the *mmap()* system call.

Table 3 lists some of the major differences of *mmap* in Tera versus those in other page based systems.

The Tera memory system supports a segment oriented virtual memory. A physical segment can vary in size from 1K to 32M 8-byte words. Since a file's logical data blocks are not guaranteed to be physically contiguous within the buffer cache. continuous virtual addresses cannot be ensured across data block boundaries. Therefore, for simplicity our current implementation of *mmap()* with *PROT_SHARED* flag allows at most one file block of data to be mapped at any one time to a segment.

In addition, the Tera's physical memory is separated into program and data memory. Program memory is not writeable by a user level thread. Therefore a protection of *PROT_EXEC* for *mmap()* has been eliminated from the Tera specification.

3.3 Extensions for Tera

According to SVID 3. *mmap()* cannot write beyond an existing end of file. On Tera, we have extended *mmap()* to map to newly created data blocks for files that are opened with write permission. SVID 3 specifies that the protection option of *PROT_WRITE* is defined as *PROT_READ and PROT_WRITE*. However, Tera provides write-only access to memory. Therefore, the protection option of *PROT_WRITE* is defined as write-only.

To allow a user to have better control over the actual mapping of one file block at a time we have extended the *mmap()* semantics by creating a new system call, known as *mmap_fsblk()*.

```
pa = caddr_t mmap_fsblk(
             caddr_t addr,
             int *len,
             int prot.
             int flags.
             int fd.
             off_t   off);
```

Page Based Systems	Tera
Page oriented	Segment oriented
Execute in data pages	No execute in data segments
Map across fixed phys page boundary	No map across variable phys segment boundary

TABLE 3: Compare Mmap on Tera vs on Page Based Systems

The major difference between *mmap()* and *mmap_fsblk()* is the parameter *len*. In *mmap_fsblk()*, *len* is a pointer to return in bytes the actual size of the segment mapped. *Len* is calculated as follows:

```
    int lbkno = (off + f_blksize) / f_blksize;
    int *len  = (lblkno * f_blksize - off);
```

where:

```
    lblkno is the logical block number where offset (off) lies
           and ranges from 1 on.
    f_blksize is the data block size of the file
```

References

[1] Tera Computer Company. *TERA Principles of Operation*. 1992.